A Certified Compiler for Verifiable Computing

Cédric Fournet, Chantal Keller, Vincent Laporte

A Certified Compiler for Verifiable Computing

Cédric Fournet
Microsoft Research
fournet@microsoft.com

Chantal Keller
LRI, Univ. Paris-Sud, France
(CNRS, CentraleSupelec, Université Paris-Saclay)
Chantal.Keller@lri.fr

Vincent Laporte
Université Rennes 1, France
IMDEA Software Institute, Spain
vlaporte@imdea.org

***Abstract*—In cryptology, verifiable computing aims at verifying the remote execution of a program on an untrusted machine, based on its I/O and constant-sized evidence collected during its execution. Recent cryptographic schemes and compilers enable practical verifiable computations for some programs written in C, but their soundness with regards to C semantics remains informal and poorly understood.**

We present the first certified, semantics-preserving compiler for verifiable computing. Based on CompCert and developed in Coq, our compiler targets an architecture whose instructions consist solely of quadratic equations over a large finite field, amenable to succinct verification using the Pinocchio cryptographic scheme. We explain how to encode the integer operations of a C program first to quadratic equations, then to a single cryptographically-checkable polynomial test. We formally prove that, when compilation succeeds, there is a correct execution of the source program for any I/O that pass this test. We link our compiler to the Pinocchio cryptographic runtime, and report experimental results as we compile, run, and verify the execution of sample C programs.

I. Introduction

In cryptology, *verifiable computing* aims at safely outsourcing the execution of a program to an untrusted machine:

- given some inputs, the *worker* executes the program and accumulates cryptographic evidence; it then returns the resulting program outputs and evidence;
- given some inputs, outputs, and evidence, the *verifier* checks their consistency to confirm that those I/O indeed reflect a correct execution of that program.

We expect the verification process to be simpler and more efficient than the original computation. (Otherwise, the verifier would just discard the evidence and re-execute the program.) Besides verifier performance, another motivation may be that the verifier does not have access to data available to the worker, inasmuch as that data is deemed private, or is too big. On the other hand, verifiable computing often incurs a large overhead for the worker to produce evidence.

Novel cryptographic schemes enable the non-interactive verification of general computations in *constant time*, and even efficient implementations, interpreters, and compilers. In particular, Pinocchio [1] and its Geppetto compiler [2] take as input a C program and yield a public *verification key* such that verifying the program I/O and evidence with this key ensures that the worker correctly executed this program with these I/O. Pinocchio has been applied to various programs (e.g. for linear algebra, encryption, data processing, and even X.509 certificate-chain validation) that perform millions of operations, and yet its evidence of correct computation consists of just 288 bytes verifiable in ~ 10 milliseconds.

Pinocchio is not for all C programs: although C is a convenient language for expressing low-level integer computations, compiling for cryptographic verification comes with serious restrictions on program control and data flows. Hence, Pinocchio compilers are not meant to turn arbitrary C code into practical VC schemes. They all involve compilation to circuits, and share their fundamental limitations: computations are statically bounded; loops are unfolded; and circuitry is needed for all branches, not just those taken. (This is similar to the use of FPGAs: by unfolding selected fragments of a C program to circuits, one can significantly improve its performance, despite serious restrictions on those fragments.)

Also, as usual with cryptography, the probability that an adversary will produce false evidence reduces to the probability of solving (supposedly) hard problems; it is not null, but it can be made arbitrarily small but using larger keys. For example, Pinocchio relies on elliptic-curve assumptions; it aims at 128-bit security, that is, security against computationally bounded adversaries, able to perform much less than 2^{128} operations.

Besides cryptographic assumptions, however, the practical security of Pinocchio also crucially depends on the correctness of its compiler—why should we trust the verification key to vouch for the execution of a given source C program? This question is challenging because:

- Verifiable computing matters precisely when the worker has an incentive to produce false evidence. He may choose any execution trace and intermediate values to exploit corner cases in the compiler or the language semantics.
- The verifier is not given access to intermediate values, hence she cannot detect anomalous executions. (Pinocchio's evidence seems random; it carries no extra information about the execution chosen by the worker.)
- The compilation scheme is unusual, and involves novel encodings and optimizations: Pinocchio encodes the semantics of programs first into quadratic equations in a large prime field, then into a single polynomial divisibility test. To reduce the number of equations, it also avoids or delays conversions between field elements and machine integers. Thus, its compiler correctness depends on complex implicit representation invariants.
- Verifiable computations scale up to millions of instructions, limiting the scope of manual reviews of the resulting divisibility test to toy examples, not whole C programs.

For these reasons, static verification techniques, and in particular certified compilation, can greatly enhance trust in (dynamically) verifiable computations.

In this paper, we present PinocchioQ, a certified Pinocchio compiler from C programs (subject to the restrictions discussed above) to quadratic arithmetic systems and polynomial divisibility tests. We formally relate the success of these tests first to the existence of a solution of the compiled equation systems, then to the existence of a correct trace for their source C programs. We implement and verify our compiler as a variant of CompCert, a certified C compiler written in Coq, thereby re-using its formal semantics for C, and supplementing it with a field arithmetic back-end.

On the other hand, we do not develop a formal computational probabilistic proof for the core cryptographic constructions used by Pinocchio (taking the divisibility test to keys). Such a formal development would be of independent interest but, as opposed to the compilation process we describe, it is the focus of detailed pen-and-paper published proofs, and it applies uniformly to system of quadratic equations, irrespective of the source program that produced it. Similarly, we do not formalize the underlying elliptic-curve implementation of the field we use, whose formal verification remains beyond the state of the art (see §VIII-C for a discussion of related work). Finally, we do not address the difficult but independent problem of proving functional and security properties on source C programs.

The paper makes the following contributions:

1) a first certified compiler for verifiable computation;
2) a formalization of quadratic arithmetic programming, the encoding technique of Pinocchio, with supporting libraries for equations and Lagrange polynomials;
3) a new 'quadratic arithmetic' back-end for CompCert;
4) a soundness theorem, intuitively saying that the property checked by the cryptographic scheme—a divisibility test between formal polynomials—entails the existence of an execution of the source C program with the same I/O as those presented by the verifier;
5) an experimental evaluation, obtained by linking our compiler to the cryptographic runtime of Geppetto, illustrating certified verifiable computing on sample C programs.

More generally, we hope that our paper also sheds light on interesting programming-language semantics and compilation issues raised by modern cryptography.

Contents. We first review background materials on the Pinocchio scheme for verifiable computation, its Geppetto implementation (§II), and the CompCert compiler (§III). We present the high level architecture of PinocchioQ and explain its key aspects by example (§IV). We then describe our formal development: a certified compiler from RTL to quadratic equations (§V) and formal polynomials (§VI), and give our main theorem. We evaluate PinocchioQ on sample C programs (§VII), discuss related work, and conclude (§VIII).

Our formal Coq development and sample C programs are included in the Pinocchio/Geppetto distribution, available online at https://vc.codeplex.com/.

II. Succinctly verifying computations

We provide a definition of verifiable computation schemes, and outline the Pinocchio protocol, intuitively the cryptographic 'machine' to which we compile C programs. Compilation issues are postponed to the next sections.

In this paper, we consider *general-purpose* verifiable-computation techniques, that apply (in principle) to arbitrary algorithms, in contrast with cryptographic schemes designed for a single purpose (such as the verification of online voting or set intersection).

Complementarily, for many algorithms, it is possible to write a simpler program that only checks the correctness of a given result; one can then cryptographically verify runs of this checker, rather than a full-fledged implementation of the algorithm. Indeed, any NP algorithm can be verified in polynomial-time. Consider, for instance, outsourcing a task that involves SAT solving (see §VII for a programming example). General verifiable computing only needs to be applied to the program that produces SAT problems, checks their solutions, and outputs their relevant parts. Accordingly, the worker may first use a SAT solver as an (untrusted) oracle to find solutions, and then produce evidence of running the checker on these solutions.

A. Terminology

In cryptographic terms, the evidence we consider consists of 'zero-knowledge, universally-verifiable, succinct, non-interactive arguments of knowledge' (or SNARKs for short). Following cryptographers, and to prevent confusion with our formal development, we use 'argument' instead of 'proof' for schemes that are *computationally sound*: a computationally-limited adversary might forge an argument for a property that does not hold, albeit with a negligible probability. 'Zero-knowledge' means that the evidence does not carry any information on the instance used by the worker to build the argument. For Pinocchio, it means that the only information leaked to the verifier is the existence of a trace with those I/Os; it is achieved by embedding random factors into the evidence that cancel out in the verification process. 'Universally-verifiable' and 'non-interactive' mean that anyone, given the verification key, can verify an argument without the need to interact with the worker. 'Succinct' means that the evidence is small, in our case constant-sized.

B. Protocol description

Figure 1 outlines our verifiable computation scheme, consisting of three algorithms (*KeyGen*, *Prove*, *Verify*). The scheme is parameterized by a program p that takes inputs $\vec{x}$ and $\vec{y}$ and produces outputs $\vec{z}$. We interpret $\vec{x}, \vec{z}$ as public I/O (known by the worker and the verifier) and $\vec{y}$ as private inputs (known only by the worker). With Pinocchio, p ranges over quadratic arithmetic programs (or QAPs, for short), defined in the next subsection. For simplicity, we omit all security parameters. We refer to Gennaro *et al.* [3] and Parno *et al.* [1] for a more complete presentation.

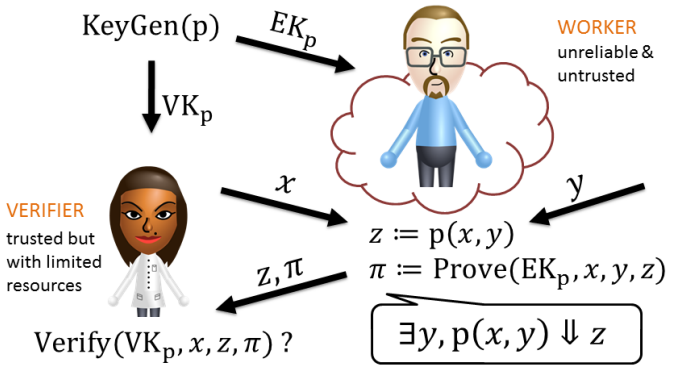


Fig. 1. Zero-knowledge verifiable computation protocol

1) $EK_p, VK_p \xleftarrow{\$} KeyGen(p)$, given a QAP p, generates public keys for evaluation and verification, respectively.
2) $\pi \xleftarrow{\$} Prove(EK_p, \vec{x}, \vec{y}, \vec{z})$, given the evaluation key, some inputs $\vec{x}$ and $\vec{y}$ and some outputs $\vec{z}$ such that $p(\vec{x}, \vec{y})$ evaluates to $\vec{z}$, produces evidence π.
3) $Verify(VK_p, \vec{x}, \vec{z}, \pi)$, given the verification key, public inputs $\vec{x}$ and outputs $\vec{z}$, and evidence π, returns a Boolean.

(The \$ indicates randomized algorithms.) Next, we give the main properties of Pinocchio:

- Functional correctness: if $p(\vec{x}, \vec{y})$ evaluates to $\vec{z}$, then $EK_p, VK_p \xleftarrow{\$} KeyGen(p)$; $\pi \xleftarrow{\$} Prove(EK_p, \vec{x}, \vec{y}, \vec{z})$; $Verify(VK_p, \vec{x}, \vec{z}, \pi)$ always returns true.
- Perfect privacy: π is statistically independent of $\vec{y}$.
- Computational knowledge soundness: given a probabilistic polynomial time adversary Adv, there exists a probabilistic polynomial-time *extractor* such that, with overwhelming probability, if $EK_p, VK_p \xleftarrow{\$} KeyGen(p); \pi, \vec{x}, \vec{z} \xleftarrow{\$} Adv(EK_p, VK_p)$; $Verify(VK_p, \vec{x}, \vec{z}, \pi)$ returns true, then the extractor on the same randomness returns $\vec{y}$ such that $p(\vec{x}, \vec{y})$ evaluates to $\vec{z}$. (See Parno *et al.* [1] for the underlying computational security assumptions.)

C. Target architecture

In Pinocchio protocols, p ranges over *quadratic arithmetic programs* (QAPs), that is, systems of equations over variables $\vec{x}$ in a field $\mathbf{F}_q$ (for a large fixed prime q), each equating a product of two linear combinations on $\vec{x}$ to a third linear combination on $\vec{x}$:

Definition 1: A *quadratic arithmetic program* $\mathcal{Q}$ of size N and degree d is defined by $3d$ vectors in $\mathbf{F}_q^N$, written $(\vec{v}_r)_{r=0..d-1}$, $(\vec{w}_r)_{r=0..d-1}$, and $(\vec{y}_r)_{r=0..d-1}$.

A vector $\vec{x} \in \mathbf{F}_q^N$ is a *solution* of $\mathcal{Q}$ when

$$\forall r \in 0..d-1, (\vec{v}_r \cdot \vec{x})(\vec{w}_r \cdot \vec{x}) = (\vec{y}_r \cdot \vec{x})$$

where $\cdot$ is the inner product on vectors in $\mathbf{F}_q^N$.

It is convenient to use affine combinations of variables rather than linear ones. To this end, we simply use a distinguished variable $x_1 \in \vec{x}$ and set $x_1 = 1$. In the following, we often omit x_1, writing e.g. $(x+1)y = z$ for $(x + x_1)y = z$.

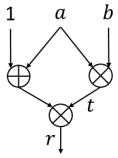
QAPs are closely related to *arithmetic circuits*, that is, circuits with additive and multiplicative gates connected by wires that carry values in $\mathbf{F}_q$: we encode each wire in the circuit as a linear combination of QAP variables, effectively inlining all additive gates, and we encode each (binary) multiplicative gate as a quadratic equation, by setting $\vec{v}$ and $\vec{w}$ to the corresponding linear combinations of their two input wires, and setting $\vec{y}$ to zeros except for a variable that holds the multiplication output. Accordingly, we will often refer to the variables $\vec{x}$ as the 'wires' of $\mathcal{Q}$. Arithmetic circuits and QAPs are 'universal', inasmuch as they can encode binary circuits (see §V-D2), but they are still very low level. As would be the case if we were compiling to hardware or FPGAs, we will need to unroll loops, and to encode dynamic memory accesses. On the other hand, each wire carries a large integer modulo q, rather than just a bit, so we can hope for efficient encodings of most operations on source C integers.

For Pinocchio provers, cryptographic processing is essentially linear in the degree of the QAP, so we can already give an idea of our programming cost model: linear operations (additions, and multiplications by constants) are free, inasmuch as they can be inlined in equations, whereas multiplications cost one equation each.

D. A first programming example

Consider the simple C program listed below (on the left) and an equivalent arithmetic circuit (on the right), with variable r for the program output.

```
int main() {
    int a = read_char();
    int b = read_char();
    write_int( (a + 1) * a * b );
    return 0;
}
```



Their semantics is captured by a QAP of degree 2, defined by two equations $ab = t$ and $(a+1)t = r$. In this particular case, assuming that the verifier checks that the two input characters fit in 8 bits (that is, $a, b \in 0..255$), none of the operations in the program overflows, and one easily checks that any solution in $\mathbf{F}_q$ for $q > 2^{32}$ is also a solution on machine integers.

E. Polynomial Encodings

Before moving on to compilation, we give an informal idea of the mechanisms used to convey a proof of knowledge of a correct valuation to a QAP in zero knowledge. These materials are used only in §VI.

Instead of separately proving (and verifying) each quadratic equation, we encode all of them into a *single* equality between high-degree formal polynomials in $\mathbf{F}_q$, and then we test this equality at a *single* random point $s \in \mathbf{F}_q$. Indeed, by the Schwartz-Zippel lemma, a non-zero polynomial of degree d evaluates to 0 at most on d roots out of q elements, and thus for $d \approx 2^{20}$ and $q \approx 2^{254}$, the probabilistic test on s wrongly accepts a non-zero polynomial with a probability well below those usually considered for computational safety.

Given $\mathcal{Q}$ of size N and degree d, we fix a formal polynomial $\delta[X]$ with distinct roots $(\alpha_r)_{r=0..d-1}$ and define $3N$ Lagrange polynomials $\vec{v}[X]$, $\vec{w}[X]$, and $\vec{y}[X]$ (one for each variable of $\mathcal{Q}$)

such that, for each $r \in 0..d-1$, we have $\vec{v}[\alpha_r] = \vec{v}_r$, and similarly for $\vec{w}[X]$ and $\vec{y}[X]$. Thus, $\vec{x}$ is a solution of $\mathcal{Q}$ if and only if, for each $r \in 0..d-1$, we have $(\vec{v}[\alpha_r] \cdot \vec{x})(\vec{w}[\alpha_r] \cdot \vec{x}) = (\vec{y}[\alpha_r] \cdot \vec{x})$ or, equivalently, there exists a polynomial $h[X]$ such that

$$(\vec{v}[X] \cdot \vec{x})(\vec{w}[X] \cdot \vec{x}) = (\vec{y}[X] \cdot \vec{x}) + h[X]\delta[X] \qquad (1)$$

Given $\vec{x}$, the worker computes the polynomials above, computes $h[X]$ as the result of a large polynomial division, and then 'only' proves that this polynomial equality holds.

To this end, Pinocchio relies on a representation of $\mathbf{F}_q$ as the points of an elliptic curve, with point additions for additions, bilinear pairings for multiplications, and a fixed generator G. (Thus, every point P is of the form $n.G$ for some $n \in 0..q-1$ but extracting n from a random P is hard.) At key generation, it samples a secret $s \in \mathbf{F}_q$ and generates keys that enable the worker (and the verifier, for the public I/O variables) to evaluate each of the polynomials above on the curve—without disclosing the value of s. Finally, using a few pairings, the verifier checks that those factors are well-formed, and multiplies them to check that (1) holds at s on the curve.

Polynomial encoding for our first example. Let $K = \{1, a, b, t, r\}$ index the 5 wires used in the QAP. We define 3 series of polynomials $(v_k)_{k\in K}$, $(w_k)_{k\in K}$, and $(y_k)_{k\in K}$ by their values at two roots (say $r_0 = 0$ and $r_1 = 1$), set to 0 except for $v_a[r_0] = 1$, $w_b[r_0] = 1$, and $y_t[r_0] = 1$ to encode the first equation, and $v_1[r_1] = 1$, $v_a[r_1] = 1$, $w_t[r_1] = 1$, and $y_r[r_1] = 1$ to encode the second equation. Let $v = \prod_{k\in K} kv_k[X]$, and similarly for w and y. We compute $v[X] = X + a$, $w[X] = b(1 - X) + tX$, and $y[X] = t(1 - X) + rX$ so the single polynomial equation $v[X].w[X] = y[X] + d[X].h[X]$ that captures the semantics of our C program becomes

$$(X + a)\,(b(1 - X) + tX) = t(1-X)+rX+X(X-1).h[X]$$

For instance, for $a = 2$, $b = 5$, $t = 10$, $r = 30$, we have $v[X] = X + 2$, $w[X] = 5 + 5X$, $y[X] = 10 + 20X$, and, after some arithmetic, we have indeed $(vw - y)[X] = 5d[X]$.

III. CompCert

CompCert [4] is a full-fledged certified compiler for the C programming language. It has been specified and proved correct using the Coq proof assistant: each intermediate language comes with a semantics, and each compilation pass comes with a theorem stating that it does not introduce new behaviors. These theorems are of the form

$$\forall p : \texttt{good}, \forall p', \texttt{Compile}(p) = \lfloor p' \rfloor \rightarrow [\![p']\!] \subseteq [\![p]\!]. \qquad (2)$$

In this statement, `good` is the set of C programs whose behavior cannot go wrong. This hypothesis is rather strong—it is not a decidable property. However, for some programs, it can be automatically discharged by static analysis [5]. The notation $\lfloor p' \rfloor$ means that the compilation returns a value p' as opposed to an error. Finally, $[\![p]\!]$ denotes the set of behaviors of a program p, explained next.

A. Execution traces

The behavior of a good terminating program is described as a set of *traces*, each trace consisting of a list of events and a final return value (see Figure 2, including definitions from `Events.v` and `Behaviors.v` in CompCert 2.4). An event is either a call to an external function, or a *volatile* memory access (load and store), or an annotation.

Global variables can be labeled in the source as volatile, to mean that the environment can provide a value for every load and observe the value of every store. Volatiles do not affect the rest of the memory (as opposed to external calls), so they are well-suited to model I/O.

Annotations are statements that can appear in the source and in any intermediate language. Their execution has no effect other than being recorded in the trace; they are thus guaranteed to be preserved across all compilation passes.

Events may record run-time values, of type `eventval`, including 32-bit machine integers (`Vint i`) and pointers. Machine integers are constructed from (mathematical) integers using `Int.repr`, and interpreted as signed (resp. unsigned) integers using `Int.signed` (resp. `Int.unsigned`).

B. Compiling and running our first example

Recall the C program given in §II. The functions `read_char` and `write_int`, along with other I/O routines, are defined in our custom header file `p8q.h`, as follows: a public input is modeled by an annotation that claims the range of the next input value, followed by a read from a volatile location; an output is modeled by a write to a volatile location. Thus, our program has behaviors of the form `Terminates tr Int.zero` for many traces `tr` of the form below (in informal syntax):

```
annot input(0, 0xFF)
vload input_public → 12
annot input(0, 0xFF)
vload input_public → 37
vstore output_run_time ← 5772
```

CompCert successively transforms the source program into intermediate representations, down to assembly code. Among these representations, we are mostly interested in RTL, for which we implement and formalize a back-end to quadratic arithmetic programs, presented in §IV and §V. RTL is a Register Transfer Language, with three-address code on infinitely many pseudo-registers [6]. For instance, our sample C program yields the following RTL code.

```
main() {
    29: x29 = annot "input"(0, 0xFF)
    27: x4 = volatile load int32 "input_public" 0()
    17: x19 = annot "input"(0, 0xFF)
    15: x3 = volatile load int32 "input_public" 0()
    9: x9 = x4 + 1
    8: x8 = x9 * x4
    7: x30 = x8 * x3
    4: volatile store int32 "output_run_time" 0(x30)
    2: x5 = 0
    1: return x5 }
```

```
Inductive event : Type :=
| Event_syscall‘(ident)‘(list eventval)‘(eventval)
| Event_vload‘(memory_chunk)‘(ident)‘(int)‘(eventval)
| Event_vstore‘(memory_chunk)‘(ident)‘(int)‘(eventval)
| Event_annot‘(ident)‘(list eventval).
Definition trace : Type := list event.
Inductive program_behavior : Type :=
| Terminates‘(trace)‘(int)
```

Fig. 2. Execution traces (omitting cases for wrong or diverging behaviors)

IV. Verifiable programming with PinocchioQ

Our certified compiler is a branch of CompCert with a new back-end that symbolically interprets RTL to generate a quadratic arithmetic program, which can then be passed to Geppetto's cryptographic engine to generate keys.

We also provide a (concrete) interpreter that runs RTL and collects intermediate values, which can then be passed as witnesses to Geppetto's engine to produce evidence, which can in turn be checked against some program I/O.

Operating at the level of RTL lets us focus on specific compilation issues, notably the efficient encoding of operations on integer registers, leaving CompCert to handle the rich syntax and semantics of standard C and perform front-end optimizations. In particular, we configure CompCert to aggressively inline code and propagate constants, inasmuch as our compiler will unfold all calls and all loops. Even so, RTL remains more concise and more convenient than, e.g., the explicit arithmetic circuits manipulated in the first Pinocchio compilers.

One central issue is the selection of arithmetic encodings, inasmuch as QAPs only add and multiply field elements. To match the 32-bit semantics of CompCert and RTL registers, we track overflows on arithmetic operations, and we switch between arithmetic and bitwise representations only to truncate their results and perform binary operations (`||`, `&&`, `<=`, ...). For instance, multiplying two registers costs one wire and one equation, but truncating the result is $64\times$ more expensive, requiring one wire and one equation for each bit of the raw result (see §V-D3). To achieve practical performance, our compiler precisely tracks integer ranges and representations, enabling us to avoid, postpone, or share most binary decompositions.

Our formal contribution has two main components, outlined below, and detailed in the next sections.

A. An abstract interpreter for RTL (§V)

The first component implements both the *compiler* and the *worker*. This modularity is achieved by interpreting RTL with two different implementations of arithmetic:

- a direct implementation, as operations in Pinocchio's finite field, for the worker to compute the output; and
- a custom implementation, as symbolic arithmetic operations, for the compiler to perform range analysis and generate quadratic equations.

This modularity also helps us separate, in the proof, the invariants of the different components. A shared state abstraction models the memory state, by providing primitives to store and load registers, to assign values, etc. The interpreter follows the control-flow of the RTL program and produces an execution trace, which is concrete in evaluation mode since all the inputs are known, and symbolic in key-generation mode, since run-time public and private inputs are not known yet. Thus, the key-generation mode produces a QAP, and the evaluation mode, a solution to this QAP.

Partial correctness of our compiler states that, roughly, for every good program (according to CompCert) such that the compiler returns a system of equations $\mathcal{Q}$ and a symbolic trace, any solution of $\mathcal{Q}$ restricted to the program I/Os yields a valid trace instance for the source program. Formally, the compiler is always allowed to fail, and it does on some programs: after inlining and partial evaluation, it will reject non-trivial pointer arithmetic and complex control flows (see §V-F). Recall that Pinocchio, its siblings, and more generally compilers to circuits are not meant to support all C programs. PinocchioQ supports their main techniques and optimizations; §VII reports its evaluation on various sample programs.

The results of our RTL interpretations (first producing a QAP, then producing its solution) can then be passed to Geppetto [2] to produce the cryptographic argument and check its validity. For functional correctness at run-time, it is essential that these symbolic and concrete interpretations be closely synchronized. In particular, whenever the compiler may apply an optimization that depends on the outcome of static analyses and affects the resulting QAP, the worker must know whether the optimization was applied, in order to collect matching evidence. (For example, it may need to know how many bits were used in the binary decomposition of an arithmetic value, in order to collect their valuation). To communicate this information, the compiler generates an additional stream of *hints*, and the worker consumes those hints to guide the collection of evidence. This hint mechanism does not interfere with the partial correctness of our compiler, but its failure may prevent the worker from producing verifiable evidence.

B. Polynomial encodings (§VI)

The second component of our formalization establishes the correctness of Pinocchio's polynomial encoding, thereby linking the conclusion of the cryptographic checks to the existence of a solution of the quadratic equations system. Given a QAP $\mathcal{Q}$, public I/Os, and cryptographic evidence, computational knowledge soundness yields (with high probability) a valuation ρ of the wires of $\mathcal{Q}$ that extends the I/Os and such that d divides $vw - y$ (reusing the formal polynomial notations of §II), and we prove that any such valuation also yields a solution of $\mathcal{Q}$.

Our two certified components are independent, as long as they agree on a representation and denotation of QAPs. Putting all the pieces together, including CompCert from C programs to RTL, we prove that, for every 'good' C program, this valuation ρ finally yields a valid source trace (§V-G). Before detailing them in the next two sections, we illustrate our compilation process on a programming example.

C. Another programming example

The C code below checks that the worker knows two non-trivial factors of some integer N. It can be seen as a simplified RSA key-generation verifier, operating on small integers for simplicity—with RSA, N would be a 2048-bit product of two secret primes.

```
#include <p8q.h>
int main(void) {
    int N = read_public_int(0, INT_MAX);
    int a = read_private_int(8);     // equations (3)
    int b = read_private_int(8);     // equations (3)
    assert (a - 1); assert (b - 1); // equations (4,5)
    assert (N == a * b);            // equations (6,7)
    return 0; }
```

At runtime, the program inputs first N, then two known factors `a` and `b` (which may be obtained by any means, e.g. by having computed N earlier as their product). N is a *public input*; in contrast, a and b are *private inputs*: the verifier cares only about their existence, whereas the worker may wish to keep them secret. The PinocchioQ header file defines their two input functions as follows:

- `read_public_int(x,y)` reads a 32-bit volatile and inserts a range annotation: PinocchioQ will assume that the read value is in range `[x,y]`. Since the input is public, the verifier will dynamically check this range condition.
- `read_private_int(k)` reads a k-bit integer. Since the input is private, the verifier cannot directly check its range condition; instead, PinocchioQ reads its k-bit binary representation, one volatile bit at a time.

The program then proceeds with three run-time assertions that $a \neq 1$, $b \neq 1$, and $N = ab$. CompCert does not natively support assertions, often used in verification-oriented code. To support them within the semantics of 'good' C programs, we encode assertion failure as non-termination: our header file defines `void assert(bool c){ if(c){} else {while(1);}}` so that any trace of a terminating execution ensures that all program assertions have succeeded.

Putting everything together: if the program has a finite trace reading N, a, and b, then the verifier can conclude that the worker indeed knows two factors of N. At compile-time, the trace is of the form

```
annot input(0, 0x7FFFFFFF)
vload input_public → x2
vload input_private → x3
...
vload input_private → x18
```

To compile our program, PinocchioQ first calls CompCert's front-end to obtain the corresponding RTL, then it symbolically interprets it and generates the QAP of size 21 (its number of wires) and degree 20 (its number of equations) listed in Fig. 3. Comments in the C code refers to equations in the figure. For instance, the RTL multiplication that evaluates the right-hand-side of the equality in the final assertion and puts the result in register x_{21} is interpreted by extending the QAP with a new wire for x_{21} and an equation relating it to the product of the values held in the registers that hold a and b.

$$x_3 \text{ to } x_{18} \text{ are bits:} \quad (1 - x_i)x_i = 0 \qquad \text{for i = 3..18} \tag{3}$$

$$a - 1 \text{ is not null:} \quad (\textstyle\sum_{i=0}^{7} 2^i x_{3+i} - 1)x_{19} = 1 \tag{4}$$

$$b - 1 \text{ is not null:} \quad (\textstyle\sum_{i=0}^{7} 2^i x_{11+i} - 1)x_{20} = 1 \tag{5}$$

$$x_{21} \text{ is the product } ab\text{:} \quad (\textstyle\sum_{i=0}^{7} 2^i x_{3+i})(\sum_{i=0}^{7} 2^i x_{11+i}) = x_{21} \tag{6}$$

$$\text{Input } x_2 \text{ equals } x_{21}\text{:} \quad x_2 \times 1 = x_{21} \tag{7}$$

Fig. 3. QAP compiled from the C program of §IV-C

Other operations may require encodings. For instance, the first two assertions are interpreted by showing that their arguments are non-null. In a field, this is equivalent to providing their inverses, hence we introduce wires x_{19} and x_{20} for those, and 'inverse' equations (4) and (5). See §V-D4 for the preconditions of this encoding.

Each private bit input (16 in total for a and b) also requires a new wire x_i. Since the worker may *a priori* pick any element of $\mathbf{F}_q$ for x_i, not just 0 or 1, we include equations $(1-x_i)x_i = 0$ to check that x_i is indeed a bit. The arithmetic representations of a and b, used in other equations, are obtained as weighted sums: for instance, a's bits are input on wires x_3 to x_{11} and a's symbolic value is the linear combination $\sum_{i=0}^{7} 2^i x_{3+i}$.

D. Implementation summary

The Coq development is available at https://vc.codeplex.com. Excluding CompCert, it has 9 kLOCs (half specifications and implementation, half proofs) supplemented by a few hundred lines of OCaml for the impure parts of the compiler (mostly reading and writing files to interoperate with Geppetto's cryptographic runtime). It comes with examples in C, illustrating the use of public and private inputs, and some meta-programming to efficiently handle verification tasks (see §VII).

V. Certified compilation

The interpreter described in §IV relies on 3 nested components:

- an abstract domain for machine integers and their operations (§V-B), with a simple, concrete implementation (§V-C) used by the worker to run the program, and a more involved, stateful, symbolic implementation that accumulates quadratic equations (§V-D) to compile the program;
- a representation of the program state (§V-E), supplementing abstract integers with pointers and integers known at compile time, and a simple model for allocating, reading, and writing memory and for pointer arithmetic;
- an RTL interpreter (§V-F), handling the program control flow and its trace.

Each component is specified so that they can be implemented and proved correct independently. Next, we explain their formalization, highlighting key aspects of the proof.

A. Monotonic State and Error Monad

We first describe a state monad (implemented in `Monads.v`) used in most of our specifications to represent computations

that depend on a state, may update it, and may also fail with an error message. A distinctive feature of this monad is that computations are *monotonic*: the final state comes with a proof that it is related to the initial state. The datatype is parameterized by a type `S` of state, a reflexive transitive relation `incr` on states, a type `E` of error, and a type `A` of returned value. Such a computation, for each initial state `s`, returns either an error `Error e` or a value `a` and a final state `s'` in relation with the initial state.

```
Context (S: Type) (incr: relation S).
Inductive result (E A: Type) (s: S) : Type :=
| Result (a: A) (s': S) `(incr s s') | Error (e: E).
Definition t (E A:Type) : Type := ∀ s, result E A s.
```

For brevity we write `STm.t A` for the datatype `t S incr E A`.

B. An abstract domain for machine integers

This domain models operations on integers; its interfaces and specifications appear in `AbInt.v`. Integer values may be unknown at compile-time, so they are represented symbolically (as values of type `c`). The domain provides basic operations such as addition or multiplication. Its implementation may be stateful—for instance, in key generation mode, it will keep track of the range of these values. Therefore each operation is wrapped in our state monad.

We give below an excerpt of its interface, with signatures for bitwise logical AND, addition, and multiplication.

```
Record arith (c: Type) : Type := {
 land: c → c → STm.t(c * (option nat * option nat));
 add: c → c → STm.t c;
 mul: c → c → STm.t (c * bool) ... }.
```

Every operator returns a result value ('variables', of type `c`); some also return *hints*, i.e., details about how the operation has been carried on. For instance, in key generation mode, a fresh equation is introduced for each multiplication in the general case, but the compiler sometimes knows that one of the operands is a constant and that no equation is needed. Such facts are then communicated to the caller through the returned Boolean. These facts are ultimately logged in a file to be used for proof generation. Similarly, bitwise operations—such as bitwise logical AND—may require a binary decomposition of theirs operands. In such cases, the returned natural numbers tell (for each operand) how many bits are introduced in this decomposition.

The specification of this domain is given with respect to a relation `solution` that provides an interpretation of the variables in the state of the domain as mathematical integers. It may also have a state invariant `Inv`.

We give below the specification for addition. Its first hypothesis states that a call to `add` on variables `x` and `y` from state σ returned as result variable `z` and updated state σ'. The second hypothesis asserts that the initial state satisfies the invariant. Then, for every solution ρ for the *final* state, the solution for variable `z` is the sum of the solutions for variables `x` and `y`. A solution is a valuation of the variables as mathematical integers, but operators of the value domain like `add` are meant to model machine arithmetic. Therefore, the values of `x`, `y`, and `z` are cast to machine integers in the conclusion (using `Int.repr`). Moreover, the resulting state still satisfies the invariant.

```
Definition add_sound (a: arith) (Inv: S → Prop)
  (solution: S → (c → Z) → Prop):
  ∀ x y σ z σ' H,
  add a x y σ = STm.Result z σ' H →
  Inv σ →
  ∀ ρ, solution σ' ρ →
  Int.add(Int.repr(ρ x))(Int.repr(ρ y)) = Int.repr(ρ z)
  ∧ Inv σ'.
```

Next, we present two implementations of abstract integers, corresponding to the two modes of the interpreter.

C. Concrete integers as elements in $\mathbf{F}_q$

The first implementation, used in proof-generation mode, represents integers as field elements (see `AbIntModp.v`). The domain is stateless and has a trivial invariant. The 'solution' of this domain is the function that maps each point to the corresponding mathematical integer between 0 and $q-1$.

This implementation is not completely trivial, since no overflows should happen. Consider the case of addition of two elements x and y in the field, seen as non-negative integers smaller than q. If their sum is larger than q, then the result does not accurately model their addition as machine integers.

The soundness of the implementation also relies on the fact that q is prime. For the specific 254-bit values of q used in the Geppetto engine, we establish this fact in Coq using the primality checker of Grégoire *et al.* [7] based on Pocklington certificates.

D. Symbolic integers and quadratic equations

The second implementation of integers is used in key-generation mode; it treats run-time values symbolically. (Its implementation is in `QAP.v`; its correctness proof is in `QAPProof.v`.)

Quadratic equations are represented by three sparse maps `cV`, `cW`, and `cY`, from equation names (a.k.a. *roots*) to symbolic values (see below). Therefore, for each root `r`, these maps define a quadratic equation whose solution ρ satisfies:

$$[\![\texttt{cV}[\texttt{r}]]\!]\rho \times [\![\texttt{cW}[\texttt{r}]]\!]\rho = [\![\texttt{cY}[\texttt{r}]]\!]\rho \tag{8}$$

where $[\![\texttt{cV}[\texttt{r}]]\!]\rho$ is the element of $\mathbf{F}_q$ obtained by evaluating the symbolic value `cV[r]` given the solution ρ.

A symbolic value is either: a linear combination of *wires* in $\mathbf{F}_q$ and a range; or a binary decomposition. These values are stored in a table in the state of the domain. The domain clients only get indices to the table, so they cannot forge values (e.g., with an incorrect range). The domain invariant ensures the consistency of all representations.

Linear combinations are by far the most efficient representation. They are modelled as partial maps from wires to coefficients, each wire being an input, an output, or an intermediate result. The (finite) set of bound wires is written $\mathcal{W}$. Among the various possible implementations of this signature, lists tend to be the most efficient one, especially in terms of memory but also in time, since these maps are extremely

sparse. (The signature of linear combinations is defined in `LinearCombination.v`; its list implementation is in `LCList.v`.) To each linear combination is associated a *range* into which it must be evaluated (see §V-D1).

Binary decompositions are required for operations, such as bitwise AND, right shift, and comparisons ($\leq$), that access individual bits of integers. They are modelled as lists of linear combinations, each representing a bit of the value.

1) Range analysis: There are three kinds of integers involved in the compilation: elements in field $\mathbf{F}_q$ (which can be seen as integers between 0 and $q-1$), mathematical integers in $\mathbf{Z}$, and 32-bit machine integers. In particular, the QAP is a system of equations over $\mathbf{F}_q$ that model computations on machine integers.

To make explicit the link between a field element, represented as a linear combination of wires, and the corresponding machine integer, each such linear combination is associated with a *range*: an interval of mathematical integers in which the given value would be if the program was operating on mathematical integers. These intervals are computed from the annotations attached to public inputs, the checks that private bitwise inputs are indeed bits, and a range analysis for all operations. (Interval arithmetic is coded in `Interval.v`.)

Given an element $x \in \mathbf{F}_q$ and an interval $[a;b] \in \mathbf{Z}^2$ that is *not too large* (that is, $b - a < q$), the *projection* of that element on this interval is (if it exists) the unique mathematical integer $y \in [a;b]$ in the interval such that x and y are congruent modulo q.

To establish the soundness of the various arithmetic operations specified in §V-B, we define (and prove) as invariant `Inv` the following properties: for every solution to the QAP, for every value in the QAP:

- if the value is a linear combination of wires with a predicted range, projecting its valuation for this solution on this range succeeds; and
- if the value is a binary representation, projecting the linear combinations representing the bits on the range $0..1$ succeeds.

The invariant follows from these properties: a valuation of the wires is a solution of the QAP only if the values of the public inputs satisfy their annotations, and hence fall in their predicted ranges; and if a computed range is *too large* (i.e. wider than q) then compilation fails. (An alternative is to trigger a truncation to 32 bits; see §V-D2.)

2) Bitwise representation and truncation: Switching from bitwise to arithmetic representation is easy: as illustrated in §IV-C, the linear combination is the sum of the weighted bits, and the range is $0..2^\ell - 1$ where ℓ is the number of bits. The other direction, however, requires to ask the worker for a *binary decomposition* of a given linear combination a, that is, to provide those bits as private inputs. We rely on range analysis to bound the number of bits that are required, by asking for the range to be included in $0..2^\ell - 1$ and then requiring ℓ bits. (We leave as future work other optimizations to minimize ℓ, notably for signed integers.) We then add to the system $\ell + 1$ equations:

$$\textstyle\bigwedge_{i=0}^{\ell-1}(1 - b_i)b_i = 0 \quad \wedge \quad 0 = a - \sum_{i=0}^{\ell-1} 2^i b_i$$

to ensure that all the b_is are bits, and that a is equal to its binary representation. (Technically, taking ℓ too small would not compromise soundness, so the guess need not be proved correct; however, bad guesses would compromise completeness, as the worker would not be able to satisfy the last equation.)

Binary decompositions are costly, and should be avoided for performance. To this end, known binary representations are cached in the QAP, using the state monad. To establish soundness of the arithmetic operations, we thus extend the QAP invariant with the property that, for any solution, the stored binary representations equal (modulo 2^{32}) the arithmetic values they represent.

Binary decomposition often operates on values whose range is wider than 2^{32}, as the result of prior arithmetic operations. To implement our source 32-bit semantics, we then discard the higher-order bits immediately after generating the last equation above. Hence, only truncated values with at most 32 bits are cached.

3) Arithmetic and bitwise operations: Arithmetic operations take linear combinations and produce a new one.

To compute the addition of two values (named by their indices `a` and `b`), we first read in the state the linear combinations `x` and `y` that they represent; then we add their linear combinations (operator `LC.add`), we add their ranges, and we generate and return a fresh index for the resulting value. The code is given below:

```
Definition add a b :=
    do (x, xr) ← get_lc a; do (y, yr) ← get_lc b;
    fresh_lc (LC.add x y) (Interval.add xr yr).
```

Multiplication is more involved. There is an easy case when one of the operands is known at compile-time. In this case, as for the addition, we perform the multiplication on the linear combinations and on the ranges. Otherwise, when both operands are arbitrary linear combinations, a fresh wire `w` is returned and a new equation `x × y = w` is stored in the QAP. A Boolean hint indicates whether a new wire has been introduced.

```
Definition mul a b :=
  do x ← get_lc a; do y ← get_lc b;
  let '(x, xr) := x in let '(y, yr) := y in
  match Interval.is_const xr with
  | Some i => do v ← fresh_lc (value_of_lc (
    LC.muln (f_of_z i) y, Interval.muln i yr));
    STm.ret (v, false)
  | None => match Interval.is_const yr with
  | Some j => (* similar case *)
  | None => let range := Interval.mul xr yr in
    do w ← fresh_wire Bank_Local;
    let res := lc_of_wire w in
    do r ← fresh_eqn x y res;
    do lc ← fresh_lc (value_of_lc (res, range));
    STm.ret (lc, true) end end.
```

Bitwise operations, on the other hand, operate on binary decompositions, and produce a new one. They trigger binary decompositions on demand (the need for such a decomposition

is passed as a hint to the worker), which are then cached for performance reasons.

We rely on the following bitwise, algebraic encodings of logical operators (using C syntax below) on single bits:

$$!a = 1 - a \qquad a \,\&\&\, b = a * b$$
$$a \,||\, b = a + b - a * b \qquad a \,\hat{}\, b = a + b - 2a * b$$

Further, bitwise logical operators (`&`, `|`, `^`) and bitwise shifts on integers (`<<`, `>>`) are coded as operations on bit lists.

The soundness of all these operations, as stated for addition in §V-B, rely on the invariant given is §V-D2.

4) Equality checks and assertions: How to implement equality checks `a == b` ? Recall that, in C, this operator returns 1 if a and b are equal, and 0 otherwise. If $a, b \in 0..1$, then we may use $1 - a - b + 2ab$. In the general case, a naive idea would be to trigger a binary decomposition of $a - b$. We rely on a more efficient encoding, requiring just two equations. Let x be the result of the test. We prove knowledge of an auxiliary private input y such that

$$(a - b)y = 1 - x \quad \wedge \quad (a - b)x = 0 \tag{9}$$

If $a = b$, then $x = 1$ by the first equation, otherwise $x = 0$ by the second equation, so the encoding is correct. Its completeness is more subtle, as we need a witness for y. If $a = b$, then we set $y = 0$. Otherwise, we compute $y = (a - b)^{-1}$ in $\mathbf{F}_q$.

Another difficulty is to determine conditions on the range of a and b. Indeed, we must interpret $a = b$ as equality modulo 2^{32}, whereas the encoding above establishes equality modulo q. For example, consider the program fragment

```
unsigned int a = 3;
unsigned int b = 1422342341 + 2872624958;
assert(a != b);
```

According to the C semantics, the assert fails since the addition on the right-hand-side of b yields $2^{32} + 3$, so this program has no valid trace. In this case, C compilers may issue a warning, but more generally a and b may be the result of complex intermediate computations, depending on values provided by the worker. Using the encoding above, 2^{32} does have an inverse in $\mathbf{F}_q$ so, without some cautious management of integer ranges at compile time, we would let the worker prove that the program actually has a trace.

We use the same idea to encode assertions `assert(g)`. To ensure that the guard `g` is not null, we use an equation similar to (9, left), namely `g` $\times y = 1$ where y is a fresh wire. Any solution proves that `g` is not null in $\mathbf{F}_q$, so we also check that there is at most one null value in the predicted range of `g`.

E. State abstraction

This layer models the RTL state, i.e., its memory and registers contents. (Its specification is given in `AbState.v`.) It is parameterized by an instance of the value domain that operates on variables of some type `c`. It uses a dedicated expression language (of type `expr`, defined in `Expr.v`), with loads and pointer arithmetic, parameterized by the type of its variables.

It features several operators, including `sa_assign(r, e)` which models the assignment to register `r` of the result of the local computation described by the expression `e`; `sa_eval(e)` which forces the evaluation of expression `e` to a concrete value; `sa_assert(e)` which takes into account that expression `e` is kown to evaluate to a true value in the current state.

Its soundness is stated with respect to a relation `mi` between abstract and concrete states, at a particular solution. The proposition `mi` ρ σ `sp rs m` reads as follows: abstract state σ and concrete state `(sp, rs, m)` (where `sp` is the stack pointer, `rs` the register state, and `m` the memory state) are related at solution ρ. The actual meaning of *being related* is not relevant to specify a sound abstraction; however, a particular implementation will be proved sound with respect to a particular instance of this `mi` relation.

Each operator of the `state_abstraction` record comes with a soundness property; for instance, the `sa_assign` operator has the following specification.

```
Definition sa_assign_sound : ∀ dst e σ σ' H,
  sa_assign abState dst e σ = STm.Result tt σ' H →
  ∀ ρ, solution σ' ρ →
  ∀ sp rs m, mi ρ σ sp rs m →
  ∃ v,
  rtl_eval_expr ge sp rs m e tt = STm.Result v tt I
  ∧ mi ρ σ' sp (rs # dst <- v) m.
```

Its first hypothesis states that evaluation of the `sa_assign` operator to a destination register `dst`, expression `e` and state σ returned updated state σ'. The soundness condition states that, for any solution ρ to the *final* state σ', and any concrete state `(sp, rs, m)` related to the initial state σ at ρ, expression `e` evaluates in the concrete state to some value `v` and the updated concrete state (after assigning `v` to the destination register) is related to the final abstract state.

This domain communicates with the impure part of the compiler, through a dedicated state monad, called *oracle*. This monadic interface features operators to implement inputs, outputs and hint events. This interface has no specification: the compiler makes no hypothesis about its behavior.

The implementation of this domain (in `AbStateImpl.v` and `AbStateImplProof.v`) embeds the oracle, the value domain, and a memory—implemented as a map from concrete addresses to symbolic values—to implement loads and stores. The `mi` relation used to prove the soundness of this implementation states that all values in the memory actually reflect the values at the same addresses in the RTL memory, and that the invariant of the value domain is satisfied.

F. RTL interpreter

Finally, the interpreter follows the control-flow of the source program and produces an execution trace (files `Run.v` and `RunProof.v`). Since interpretation may take place at compile-time, run-time values may be unknown; therefore the interpreter is parameterized by a type `var` of symbolic names for values.

The interpreter only provides partial support for conditionals: it forces the concrete evaluation of the guard (calling `sa_eval`) to decide which branch to take and, if the Boolean value of

this expression cannot be computed (at compile time), it only support the two patterns below.

Assertions are defined in C as conditional infinite loops, and interpreted by calling the `sa_assert` function of the state abstraction, explained in §V-D4.

Conditional assignments such as `x = e;` guarded by a Boolean condition `b` are interpreted as an equivalent, *unconditional* assignment `x = (e - x)*b + x;` provided `x` is initialized and `e` has no side effect. This enables us to compile, for instance, the program listed below, which takes as inputs a, b, N and repeatedly adds b to a, with N setting the actual number of iteration. At compile-time, an upper bound P on N is provided, leading to a QAP of degree linear in P.

```
int32_t P = read_compile_time();
int32_t N = read_private_int(31);
assert ( N <= P );
int32_t a = read_public_int(0, INT_MAX);
int32_t b = read_public_int(0, INT_MAX);
for( int i = 1 ; i != P ; ++i )
   if ( i <= N ) a += b;
```

As was the case for C and RTL, our backend comes with a notion of trace that records annotations and I/O on volatiles. Given a valuation ρ of the symbolic `var` as CompCert's run-time values, a trace `r` of the interpreter, and a symbolic return value `v`, we can compute a corresponding concrete run-time behavior, written `concrete_behavior` ρ `r v`. Note that the valuation ρ is not directly a solution to a QAP. Given a QAP solution, we first evaluate QAP expressions to values in $\mathbf{F}_q$; then, relying on ranges, we project them to mathematical integers; and finally we truncate them to 32-bit integers.

G. Main theorem (formally)

Our correctness theorem (in `QapGenProof.v`) is given below:

```
   Theorem qap_gen_correct :
 1:  ∀ (prog: Csyntax.program),
 2:    good prog →
 3:  ∀ (fuel: nat)
 4:    (tr: atrace val)
 5:    (v: val)
 6:    (σ: QAP.cqap (zp large_prime)),
 7:    qap_gen prog fuel = Result (tr, v, σ) →
 8:  ∀ (ρ_IO: 𝒲 → Z),
 9:    (∃ ρ, extends σ ρ_IO ρ ∧ cqap_solution σ ρ) →
10:    (∃ ρ', extends' σ ρ_IO ρ' ∧
11:           concrete_behavior ρ' tr v ∈ ⟦ prog ⟧)
```

The 'goodness' hypothesis (line 2) states that the source program cannot go wrong, according to the CompCert C semantics (see §III).

The 'static' hypothesis (line 7) states that the compiler successfully returned a symbolic trace `tr`, a symbolic final value `v`, and a final QAP σ. (The `fuel` argument is a technicality to ensure that the compiler always terminates: compilation fails when it runs out of fuel.)

The 'dynamic' hypothesis (line 9) states the existence of a solution ρ to the QAP that coincides with (`extends`) the inputs and outputs recorded in the valuation ρ_{IO}: the `cqap_solution` relation means that all equations and all range conditions on the public input values are satisfied.

The conclusion states the existence of a terminating execution of the source program, with a trace characterized by ρ_{IO}, `tr`, and `v`, as follows. This execution uses a valuation ρ' (from variables to machine integers) that coincides with ρ_{IO} on the input and output variables (line 10) and that also provides witnesses for the intermediate program variables. This valuation is used to evaluate the symbolic values in `tr` and `v` in order to build the concrete trace (line 11). The proof relies on CompCert correctness theorem (equation (2) in §III), specialized to the front-end compiler from C to RTL (in `RTLComplements.v`) composed with the correctness of our symbolic RTL interpreter.

VI. Polynomial encoding and testing

So far, we have shown that any solution to the QAP compiled by PinocchioQ ensures the existence of a trace of the source program with matching I/Os (§V) and explained that the knowledge of such a solution can be further reduced to a polynomial divisibility test, which can be probabilistically checked using an elliptic-curve encoding (§II). We wrap up our formal development by showing that the divisibility test indeed also ensures the existence of this trace. (As explained in §I, the computational soundness of the Pinocchio's cryptographic check is outside the scope of this work; its formal proof would require tools for probabilistic, polynomial-time reasoning, such as EasyCrypt [8].)

Using the notations of §II, given a QAP $\mathcal{Q}$, assume knowledge of a valuation $\rho : \mathcal{W} \to \mathbf{F}_q$ such that $\delta[X]$ divides $v[X]w[X] - y[X]$, where $v[X]$, $w[X]$ and $y[X]$ are the Lagrange polynomials defined by $\mathcal{Q}$ and ρ. From these assumptions, we establish that ρ is a solution of $\mathcal{Q}$, in the sense of §V-D.

The core of this formalization is a construction of $v[X]$, $w[X]$ and $y[X]$, via a new library of Lagrange polynomials. It relies on the SSReflect [9] and MathComp [10] libraries for constructive arithmetic and algebraic reasoning in Coq. It uses their support for finite fields, polynomials, and big operators.

Lagrange interpolation. (See `verifier/Lagrange.v`.) Given ℓ points in the plane whose first components are pairwise distinct, our library constructs their Lagrange polynomial and establishes its three main properties:

(a) it fits the points;
(b) it has degree at most ℓ; and
(c) it is the unique polynomial with these 2 properties.

Polynomial encoding. We now suppose the existence of a valuation ρ of the wires of some QAP $\mathcal{Q}$. Recall that ρ determines the values of $v[X]$, $w[X]$ and $y[X]$ at the roots α_r of $\mathcal{Q}$; these three polynomials can thus be easily constructed using our library for Lagrange polynomials.

Moreover, we also construct the divisor polynomial $\delta[X]$ as the product of monomials $X - \alpha_r$ for all roots α_r in $\mathcal{Q}$.

Solution to the QAP. From this construction and from the divisibility assumption (that comes from the cryptographic check) stating that $\delta[X]$ divides $v[X]w[X] - y[X]$, we can deduce, by standard reasoning on polynomial roots, that ρ is a solution to $\mathcal{Q}$.

Putting everything together. The formalization presented in this section can be summed up as:

```
Theorem verifier_correct (σ: QAP) (ρ_IO: 𝒲 → Z) :
    check_range σ ρ_IO →
    ∀ ρ, extends σ ρ_IO ρ →
    dvdp (polyD σ) (polyQ σ ρ) →
    cqap_solution σ ρ.
```

where:

- `check_range` checks that the interpretation of the input and output variables given by ρ_{IO} satisfies the ranges declared by the programmer (see §IV-C), and
- `dvdp (polyD σ)(polyQ σρ)` is the divisibility assumption, subject to cryptographic testing.

Composing this result with the main theorem presented in §V-G, we obtain our final theorem, listed below, relating the polynomial post-condition of cryptographic proof verification to the existence of a finite trace of the source C program.

```
Theorem pinocchioq_correct :
  ∀ (prog: Csyntax.program)
    good prog →
  ∀ (fuel: nat)
    (tr: atrace val)
    (v: val)
    (σ: QAP.cqap (zp large_prime)),
    qap_gen prog fuel = Result (tr, v, σ) →
  ∀ (ρ_IO: 𝒲 → Z),
    check_range σ ρ_IO →
    (∃ ρ, extends σ ρ_IO ρ ∧
          dvdp(polyD σ)(polyQ σ ρ)) →
    (∃ ρ', extends' σ ρ_IO ρ' ∧
          concrete_behavior ρ' tr v ∈ ⟦ prog ⟧).
```

VII. Evaluation

We evaluated the functionality and performance of our certified compiler on a series of C programs, summarized below. Using the Verasco static analyzer of Jourdan *et al.* [5], we have also confirmed that all these programs have a well-defined semantics, as defined by CompCert.

Our experimental results are gathered in Table I. For each source program, the table gives the number of RTL instructions in the program both statically (after CompCert's front-end) and dynamically (after PinocchioQ's loop unrolling); the size (number of wires) and degree (number of equations) of the generated QAPs; and the times to compile from RTL to QAP; to interpret the program and generate all intermediate wire values; to generate keys given the QAP; to generate evidence given the keys and a QAP solution; and to verify this evidence. These last three numbers rely on Geppetto's cryptographic engine, extended to accept QAPs and valuations produced by PinocchioQ.

Timing measurements have been performed on a Linux desktop with an Intel® Core™ i7-3520M CPU @ 3.6 GHz and 8 GiB of DDR3 RAM. Each number is the average execution time of five runs (standard deviation is low). The *partial* columns present the times spent in the RTL interpreter only, whereas the *total* columns present the whole running times including the CompCert front-end (common to both modes, performing some optimizations as constant-propagation and common-subexpression-elimination) and the final printing of the outputs (serialization of large QAPs is costly).

First and **Factorization** are our running examples.

Bachet takes N as public input and checks that it equals the sum of four squares, taken as private input. Bachet's theorem guarantees the existence of such a decomposition for all positive integers, which is sometimes used to prove that N is positive in zero-knowledge. (PinocchioQ relies on a more efficient binary decomposition; see §V-D2.)

Matrix takes three $n \times n$ matrices A, B, and X as input, for some n fixed at compile-time. (We use $n = 10$ and $n = 100$.) The matrices A and B are public, while X is private to the worker. The program checks the equation $AX = B$, thereby implementing verifiable matrix division.

SHA takes a message as input: a bytestring of fixed length (4, 96, or 159) and returns its SHA1 cryptographic hash; its main loop performs 80 iterations mixing bitwise and arithmetic operations, thereby testing ranges and binary decompositions. (We also experimented with SHA256.)

SAT is a family of C programs, each checking that a particular Boolean formula in conjunctive normal form (CNF) is satisfiable, by taking a Boolean assignment as private input and evaluating the formula. It illustrates the efficient verification of an NP problem, and the use of an application-specific encoding in C, before calling PinocchioQ, discussed in the introduction.

We wrote an auxiliary translation from CNF formulas to C, as follows. Each variable is coded at a private input bit; the negation of x is coded as $1 - x$. Each clause is coded as the sum of all their literals. Finally, the whole formula is the multiplication of all its clauses, and the program asserts that it is not null. The encoding is satisfiable if there exists a valuation of its integers in $0..1$ that makes the resulting multiplication non-zero. We formally proved in Coq that our translation is sound: if the encoding is satisfiable, then the initial formula is satisfiable (file `SatCoding.v`). The encoding is not generally complete, since (in principle) a 32-bit overflow might nullify the product.

The worker expects as private input a valuation that solves the SAT instance. The cryptographic argument justifies that the encoding is non-null hence, by correctness, that the original formula is SAT and that the worker knows a solution.

We evaluate our compiler on particular SAT instances based on the Boolean encoding of a variant of the pigeonhole principle, stating that we want to put n items into exactly n containers such that no container contains more than one item. We report results for $n = 20$ and $n = 50$, respectively. For instance, for $n = 20$, the CNF has 3820 disjunctions on 400 variables, and hence the resulting QAP has 4220 equations.

Experimental Validation. These benchmarks are typical C programs used for evaluating VCs. In particular, Matrix and SHA1 are Geppetto's sample programs, modified to fit our use of volatiles for I/O, and thus provide a fair basis for comparison. Certified compilation is slower than with Geppetto, a more complicated and imperative compiler, but otherwise the two compilers yield QAPs of similar degrees, and thus yield similar cryptographic performance. As expected with Pinocchio

TABLE I
EXPERIMENTAL RESULTS

Case	RTL instructions		Size (#wire)		Degree	Compile (s)		Evaluate (s)		KeyGen	Prove	Verify
	static	dynamic	I/O	private		partial	total	partial	total	(s)	(s)	(ms)
First	21	34	4	2	3	0.00	0.00	0.00	0.00	0.03	0.01	10
Factorization	40	167	2	19	20	0.00	0.01	0.00	0.01	0.04	0.01	12
Bachet	63	527	2	64	65	0.00	0.02	0.00	0.01	0.08	0.02	12
Matrix (10)	97	37 892	201	1800	1900	0.22	0.37	0.15	0.18	0.90	0.34	13
Matrix (100)	97	17 251 542	20 001	1 080 000	1 090 000	143.72	178.62	66.34	68.37	229.37	247.70	228
SHA1 (4)	180	29 313	7	36 138	37 082	4.59	5.65	0.14	0.28	25.82	49.75	11
SHA1 (96)	180	58 831	30	77 004	78 925	10.02	12.26	0.29	0.45	26.33	32.98	11
SHA1 (159)	180	88 251	46	116 325	119 209	15.33	18.55	0.43	0.62	32.56	49.83	11
SAT (20)	39 462	40 262	1	4220	4220	0.22	9.27	0.72	9.01	1.46	1.25	14
SAT (50)	583 902	588 902	1	63 800	63 800	5.67	1843.5	12.42	1842.9	14.07	18.81	12

schemes and already observed in their earlier implementations, worker costs grow linearly with the degree on the QAP, whereas verifier costs include 10 ms for pairings and then grow linearly with the size of the I/O (only noticeable here on matrices).

VIII. Related Work and Conclusion

A. Pinocchio

Parno *et al.* [1] presents a first practical compiler and runtime system for verifiable computations, enabled by the non-PCP scheme of Gennaro *et al.* [3]. Their Python compiler translates a subset of C similar to ours to arithmetic circuits supplemented with special gates e.g. for binary decompositions. They show that, at least for some sample code, cryptographic verification is indeed faster than recomputation. Many variants of their cryptographic schemes now exist, e.g., Danezis *et al.* [11] present a minimalistic ZK-SNARK with most succinct proofs for binary circuits coded as 'square programs', i.e. using equations of the form $(\vec{v} \cdot \vec{x})^2 = 1$. Several efficient implementations are also available [12], [13], [2], exploring, e.g., advanced encodings for random memory access and cryptographic processing (not supported by PinocchioQ). Some of these first compile programs to a custom instruction set, and carefully program QAP interpreters for those instructions [14], [15], [16], [17]. Although the QAP techniques we use are largely folklore, to our knowledge we are the first to formally study their correctness with regards to the source-program semantics.

Geppetto [2] is a recent compiler and engine for Pinocchio that further reduces the worker overhead while increasing its flexibility, via the notions of MultiQAPs and bounded cryptographic bootstrapping. It has been applied to larger examples, such as the parsing and validation of complex X.509 certificate chains for TLS and Helios. By analogy, it is to PinocchioQ what `gcc -O3` is to CompCert. PinocchioQ is based on ideas from Geppetto—similarly, Geppetto uses clang as front-end compiler, and symbolically interprets LLVM code both for compiling and for executing verifiable programs. PinocchioQ does not support MultiQAPs, bootstrapping, and some advanced optimizations, but is nonetheless a realistic compiler with a similar performance profile for the examples of §VII. Conversely, our formal development led to a better understanding of subtleties and side conditions in QAP encodings, uncovering soundness bugs in early versions of the Geppetto compiler (e.g. its equality check was not correct in case of overflows, see §V-D4). PinocchioQ also relies on their optimized cryptographic engine to generate keys, and to produce and verify evidence.

B. Cryptographic compilers and certification

Other parametric cryptographic schemes have been turned into compilers, sometimes supporting low-level programs coded in C, and producing various kinds of specialized circuits. For example, Zahur and Evans [18] propose to keep the actual circuits implicit, as we do. SPDZ [19] and its implementations also rely on compilation to arithmetic circuits, rather than binary circuits, to achieve fast multiparty computations.

A few of these compilers have been formally proved sound. Almeida *et al.* [20], [21] developed a framework to verify C implementation of cryptographic algorithms and automatically generate optimized assembly code that retains the security properties established at the C level; this work is also based on CompCert. More closely related to zero-knowledge proof-of-knowledge protocols, Almeida *et al.* [22], [23] developed ZKCrypt, a certified compiler for a subset of Σ-protocols. Seen as a zero-knowledge scheme, PinocchioQ is more succinct and general, but less efficient on simple statements.

Translation validation [24] provides another approach to certification: instead of certifying the compiler once and for all, an untrusted compiler returns certificates that can be independently checked to validate its results, one program at a time. This approach is useful inasmuch as the certificate checker is simpler to prove correct than the full compiler, but we are not aware of any convenient certificate format when compiling to quadratic arithmetic programs.

C. Verification of cryptographic primitives

PinocchioQ does not address the verification of Pinocchio's elliptic curve algorithms and their Geppetto implementations, a separate hard problem. Several recent works formalize fast multiplication algorithms on selected curves [25], [26]. To

our knowledge, however, there is no formalization of pairing algorithms or their implementations, also required by Pinocchio.

Other verification results may be applied to certify C implementations of cryptographic functionalities compiled by PinocchioQ; for instance, Appel [27] formalizes an implementation of SHA1 in C, similar to the one we compile and evaluate in §VII.

D. Conclusion

We presented the first certified compiler for general verifiable computing, from C code with diverse operations on integers down to a polynomial test efficiently enforced by cryptography. Based on Pinocchio and CompCert, PinocchioQ compiles programs and enables one to prove and verify their executions, as prescribed by the C semantics.

ACKNOWLEDGMENT

The authors would like to thank Bryan Parno and Markulf Kohlweiss for insightful discussions, and the anonymous reviewers for their comments.

REFERENCES

[1] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly Practical Verifiable Computation," in *IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, pp. 238–252.

[2] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[3] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *EUROCRYPT*, 2013.

[4] X. Leroy, "Formal certification of a compiler back-end or: Programming a compiler with a proof assistant," *POPL*, 2006.

[5] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, "A formally-verified C static analyzer," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL'15. New York, NY, USA: ACM, 2015, pp. 247–259.

[6] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[7] B. Grégoire, L. Théry, and B. Werner, "A computational approach to pocklington certificates in type theory," in *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, ser. Lecture Notes in Computer Science, M. Hagiya and P. Wadler, Eds., vol. 3945. Springer, 2006, pp. 97–113.

[8] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub, "Easycrypt: A tutorial," in *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, ser. Lecture Notes in Computer Science, A. Aldini, J. Lopez, and F. Martinelli, Eds., vol. 8604. Springer, 2013, pp. 146–166.

[9] G. Gonthier and A. Mahboubi, "A small scale reflection extension for the Coq system," Inria, Tech. Rep., 2008.

[10] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry, "A machine-checked proof of the odd order theorem," in *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, ser. Lecture Notes in Computer Science, S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., vol. 7998. Springer, 2013, pp. 163–179.

[11] G. Danezis, C. Fournet, J. Groth, and M. Kohlweiss, "Square span programs with applications to succinct NIZK arguments," Cryptology ePrint Archive, Report 2014/718, 2014.

[12] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *Proc. of the ACM SOSP*, 2013.

[13] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *Proceedings of the ISOC NDSS*, Feb. 2015.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer, "Fast reductions from RAMs to delegatable succinct constraint satisfaction problems," in *Innovations in Theoretical Computer Science (ITCS)*, Jan. 2013.

[15] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," in *Proc. of CRYPTO*, 2014.

[16] ——, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *Proc. of USENIX Security*, 2014.

[17] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Proc. of CRYPTO*, 2013.

[18] S. Zahur and D. Evans, "Circuit structures for improving efficiency of security and privacy tools," in *Proc. of the IEEE Symposium on Security and Privacy*, May 2013.

[19] I. Damgard, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits," Cryptology ePrint Archive, Report 2012/642, 2012.

[20] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 1217–1230.

[21] ——, "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations," *IACR Cryptology ePrint Archive*, vol. 2013, p. 316, 2013.

[22] J. B. Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Z. Béguelin, "Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 488–500.

[23] ——, "Full proof cryptography: Verifiable compilation of efficient zero-knowledge protocols," *IACR Cryptology ePrint Archive*, vol. 2012, p. 258, 2012.

[24] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '98. Springer-Verlag, 1998, pp. 151–166.

[25] Y. Chen, C. Hsu, H. Lin, P. Schwabe, M. Tsai, B. Wang, B. Yang, and S. Yang, "Verifying curve25519 software," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 299–309.

[26] E. Bartzia and P. Strub, "A formal library for elliptic curves in the Coq proof assistant," in *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, ser. Lecture Notes in Computer Science, G. Klein and R. Gamboa, Eds., vol. 8558. Springer, 2014, pp. 77–92.

[27] A. W. Appel, "Verification of a cryptographic primitive: SHA-256," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, p. 7, 2015.